Из вводной части вы уже знаете, что основу ООП составляют классы и их объекты. Поэтому мы с вами начнем с порядка объявления классов на языке C++ и способов создания их объектов.

Исторически так сложилось, что, фактически, классы в C++ можно объявить двумя способами. Первый состоит в объявлении структур. Например, очень простую структуру Point для представления точек на плоскости можно записать следующим образом:

```language-cpp
struct Point {
    int x, y;
};
```

Напомню, что имя структуры Point в C++ воспринимается как самостоятельный тип данных и может быть использован для создания переменных этого типа. Например:

```language-cpp
Point pt;
```

или

```language-cpp
Point* ptr_pt = new Point;
```

Так вот, в концепции ООП и переменная pt, и выделенная в куче память для ptr_pt, образуют объекты типа Point. Причем *первый* объект pt размещается *в стековом фрейме*, который ограничен по объему, а **второй**, как я уже сказал, в куче, то есть, **в свободной памяти устройства**. Второй вариант создания объектов встречается наиболее часто, так как обычно в программах предполагается создание большого числа объектов, которые, к тому же, должны существовать продолжительное время и быть доступны в разных частях программы. Как только необходимость в объекте пропадает, выделенную память нужно обязательно освободить:

```language-cpp
delete ptr_pt;
```

Однако структуры – это исторически первый вариант объявления типов объектов. С появлением в C++ ООП, как новой парадигмы, классы объектов предполагается описывать с помощью ключевого слова class по аналогии со структурами. Например:

```cpp
class Point2D {
    int x, y;
};
```

И, далее, абсолютно так же можно создавать объекты этого класса:

```cpp
Point2D pt2D;
Point2D* ptr_pt2D = new Point2D;
```

В чем же разница между этими двумя способами объявлений классов? В действительности, между структурами языка C++ и его классами имеется только одно принципиальное отличие:

Все поля структуры по умолчанию являются публичными и доступны напрямую, извне. Тогда, как поля класса по умолчанию приватны и недоступны вне класса.

Покажу это на простом примере. Через объект pt структуры Point мы можем совершенно спокойно обратиться к полю x и присвоить ему какое-либо значение:

```cpp
pt.x = 1;   // ok
```

Сделать то же самое через объект класса Point2D не получится, т.к. поле x приватно, закрыто от внешнего доступа:

```cpp
pt2D.x = 1; // ошибка
```

Во всем остальном структуры и классы идентичны. Но, несмотря на это, опять же, исторически сложилось, что структуры в программах используются для описания набора данных, а классы – для описания объектов в концепции ООП. Поэтому, когда говорят о классах в C++, то подразумевают определение через ключевое слово class, а не struct.

Что такое класс в ООП?

Мы уже знаем, что в классах могут быть объявлены поля (то есть, переменные) и методы (функции-члены классов). Например:

```cpp
class Point2D {
    int x, y;
public:
    void set_coords(int a, int b)
```

```
        {x = a; y = b;}
    void get_coords(int& a, int& b)
        {a = x; b = y;}
};
```

Здесь x, y – приватные (закрытые) поля; set_coords() и get_coords() – публичные методы. В классах по умолчанию все располагается в приватной секции.

Далее, обратите внимание, что в классе Point2D переменные x, y лишь объявляются, но не размещаются в памяти. То есть, **память под x и y нигде не выделяется**, т.к. это поля будущих объектов, а класс – лишь тип данных, но не сами данные. А вот методы set_coords() и get_coords() можно воспринимать, как обычные функции, принадлежащие классу и заданные в области видимости этого класса. То есть, **имя класса Point2D помимо типа данных еще определяет и область видимости**. А раз так, то формально к функциям класса можно обращаться по синтаксису:

```language-cpp
Point2D::get_coords;
```

В будущем эта конструкция нам еще пригодится.

Что такое объект в ООП. Неявный указатель this?

Давайте теперь детальнее посмотрим, что из себя представляют объекты классов. Пусть для простоты они формируются командой:

```language-cpp
Point2D pt1, pt2, pt3;
```

В результате, каждый объект будет размещен в своей независимой области памяти (в данном случае в стековом фрейме) и будет содержать целочисленные переменные x и y. **То есть, у каждого объекта будут свои независимые переменные x и y, которые определяют координаты точки на плоскости**.

А вот методы у всех объектов общие и располагаются в классе Point2D. Но как тогда метод «понимает», с данными какого объекта он должен работать? Об этом мы с вами тоже уже говорили. Когда происходит вызов метода через объект класса, например:

```language-cpp
pt1.set_coords(1, 2);
```

то компилятор автоматически и неявно в этот метод передает **специальный указатель с именем this, который ссылается на этот объект** (в данном примере на объект pt1). При этом тип указателя this соответствует типу класса объекта. В нашем случае – Point2D*. Соответственно, через неявный указатель this выполняется доступ к переменным x, y текущего объекта pt1. Поэтому, как вариант, мы можем записать тело метода set_coords следующим образом:

```cpp
void set_coords(int a, int b)
{
    this->x = a;
    this->y = b;
}
```

Если указатель this не прописывается:

```cpp
void set_coords(int a, int b)
{
    x = a;
    y = b;
}
```

то он подразумевается при доступе к переменным объекта.

Так, благодаря неявному указателю this, единый набор методов класса может обрабатывать данные объектов, через которые они были вызваны. Именно по этой причине обычные методы вызывать напрямую из класса нельзя. Следующая команда приведет к ошибке на этапе компиляции:

```cpp
Point2D::set_coords(1, 2);
```

Метод set_coords не может быть вызван без привязки к какому-либо объекту класса Point2D, так как компилятор не может определить значение неявного указателя this. Собственно этим методы класса отличаются от обычных функций: в методы дополнительно передается указатель this.

В заключение этого занятия напомню, что когда объекты класса формируются с помощью оператора new, например, так:

```cpp
Point2D* p = new Point2D;
```

или так:

```cpp
Point2D* p = new Point2D();
```

то обращение к переменным и методам выполняется через оператор ->:

```cpp
p->set_coords(10, 20);
```

В конце, конечно же, не забываем освобождать выделенную под объект память:

```cpp
delete p;
```

Как я уже говорил, объекты классов чаще всего формируются, как раз с помощью оператора new и уничтожаются вызовом оператора delete. Поэтому дальше мы часто будем использовать этот подход к порождению объектов класса.